Chapter 3

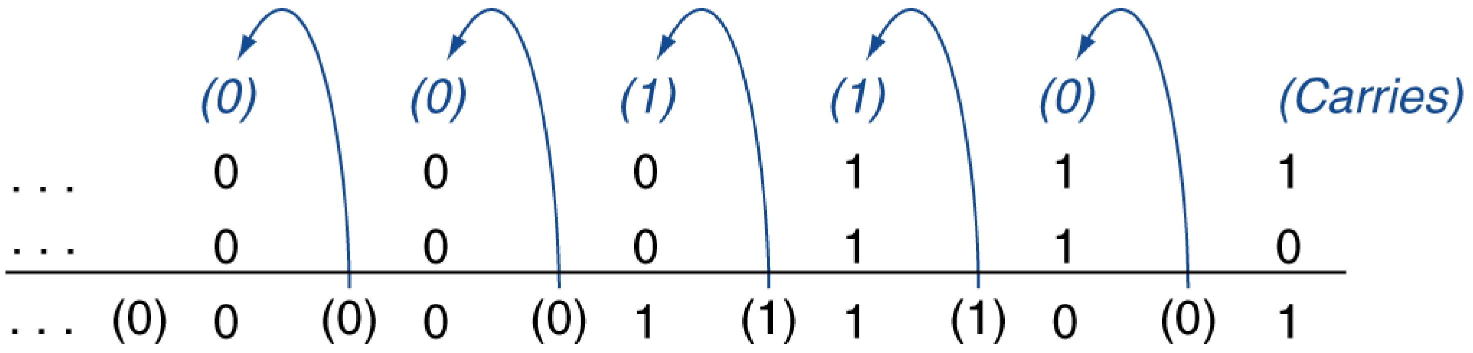
Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

```
+7:     0000 0000 … 0000 0111
–6:     1111 1111 … 1111 1010
-----------------------------
+1:     0000 0000 … 0000 0001
```

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Dealing with Overflow

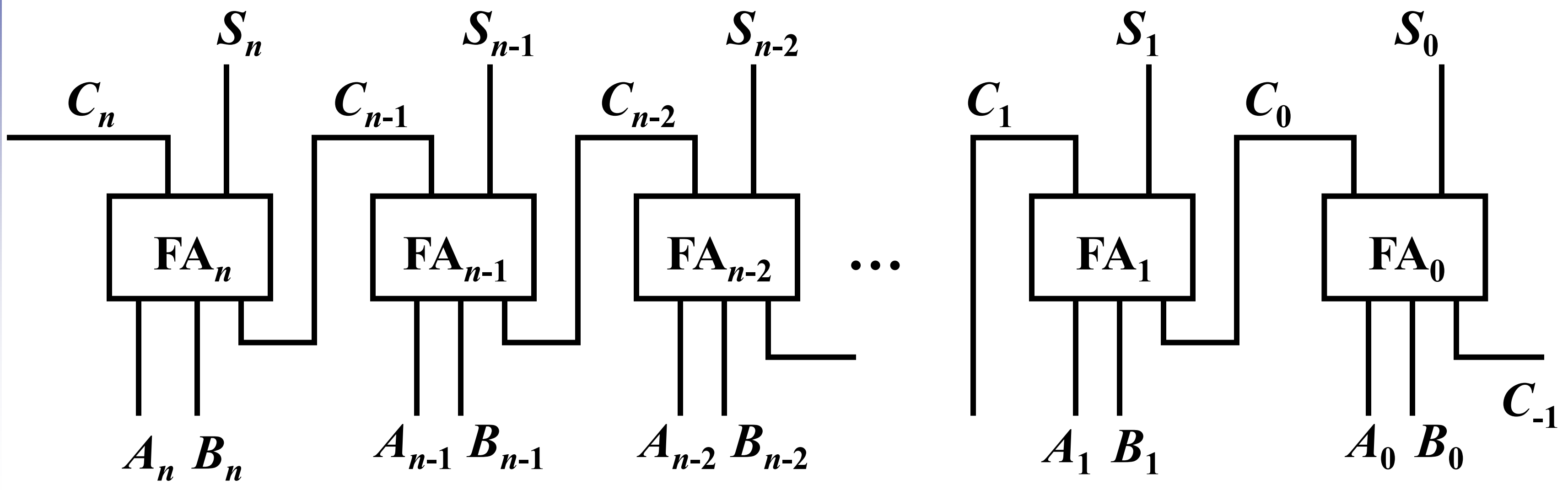
- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

二、快速进位链

1. 并行加法器



$$S_i = \overline{A}_i\,\overline{B}_i\,C_{i-1} + \overline{A}_i\,B_i\,\overline{C}_{i-1} + A_i\,\overline{B}_i\,\overline{C}_{i-1} + A_i\,B_i\,C_{i-1}$$

$$C_i = \overline{A}_i\,B_i\,C_{i-1} + A_i\,\overline{B}_i\,C_{i-1} + A_i\,B_i\,\overline{C}_{i-1} + A_i\,B_i\,C_{i-1}$$

$$= A_i\,B_i + (A_i + B_i)C_{i-1}$$

$d_i = A_i\,B_i$ 本地进位　　　$t_i = A_i + B_i$ 传送条件

则 $C_i = d_i + t_i C_{i-1}$

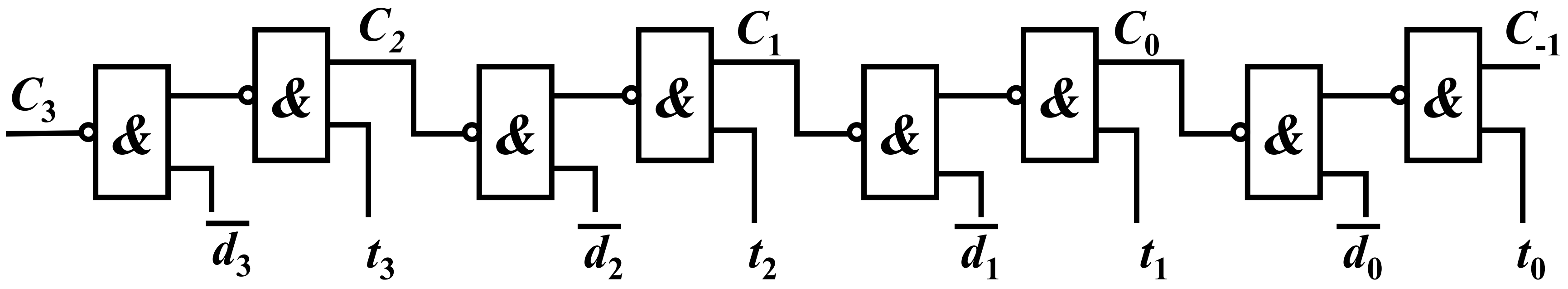
2. 串行进位链

进位链	传送进位的电路
串行进位链	进位串行传送

以 4 位全加器为例，每一位的进位表达式为

$$C_0 = d_0 + t_0C_{-1} = \overline{\overline{d_0} \cdot \overline{t_0C_{-1}}}$$
$$C_1 = d_1 + t_1C_0$$
$$C_2 = d_2 + t_2C_1$$
$$C_3 = d_3 + t_3C_2$$

设与非门的级延迟时间为t_y



4 位 全加器产生进位的全部时间为 $8t_y$

n 位全加器产生进位的全部时间为 $2nt_y$

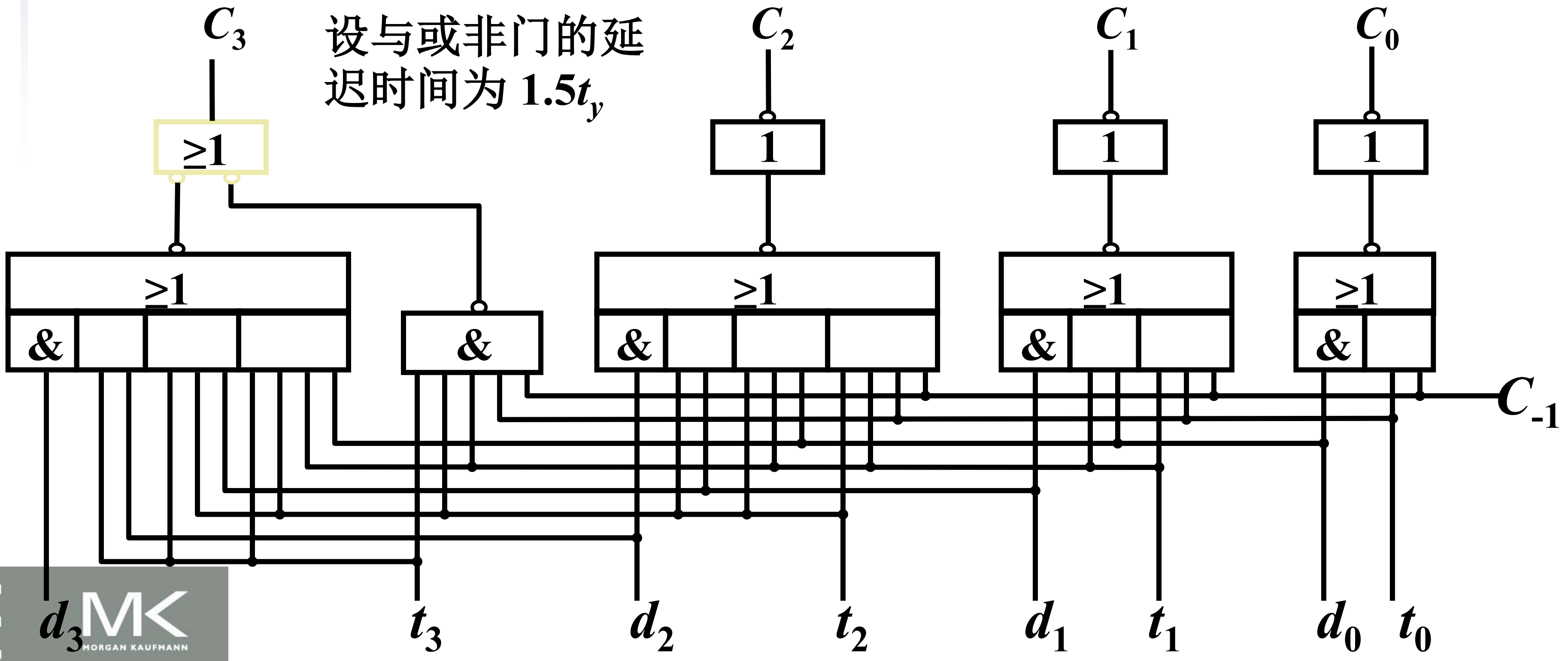
3. 并行进位链（先行进位，跳跃进位）

n 位加法器的进位同时产生　　以 4 位加法器为例

$$C_0 = d_0 + t_0 C_{-1}$$
$$C_1 = d_1 + t_1 C_0 = d_1 + t_1 d_0 + t_1 t_0 C_{-1}$$
$$C_2 = d_2 + t_2 C_1 = d_2 + t_2 d_1 + t_2 t_1 d_0 + t_2 t_1 t_0 C_{-1}$$
$$C_3 = d_3 + t_3 C_2 = d_3 + t_3 d_2 + t_3 t_2 d_1 + t_3 t_2 t_1 d_0 + t_3 t_2 t_1 t_0 C_{-1}$$

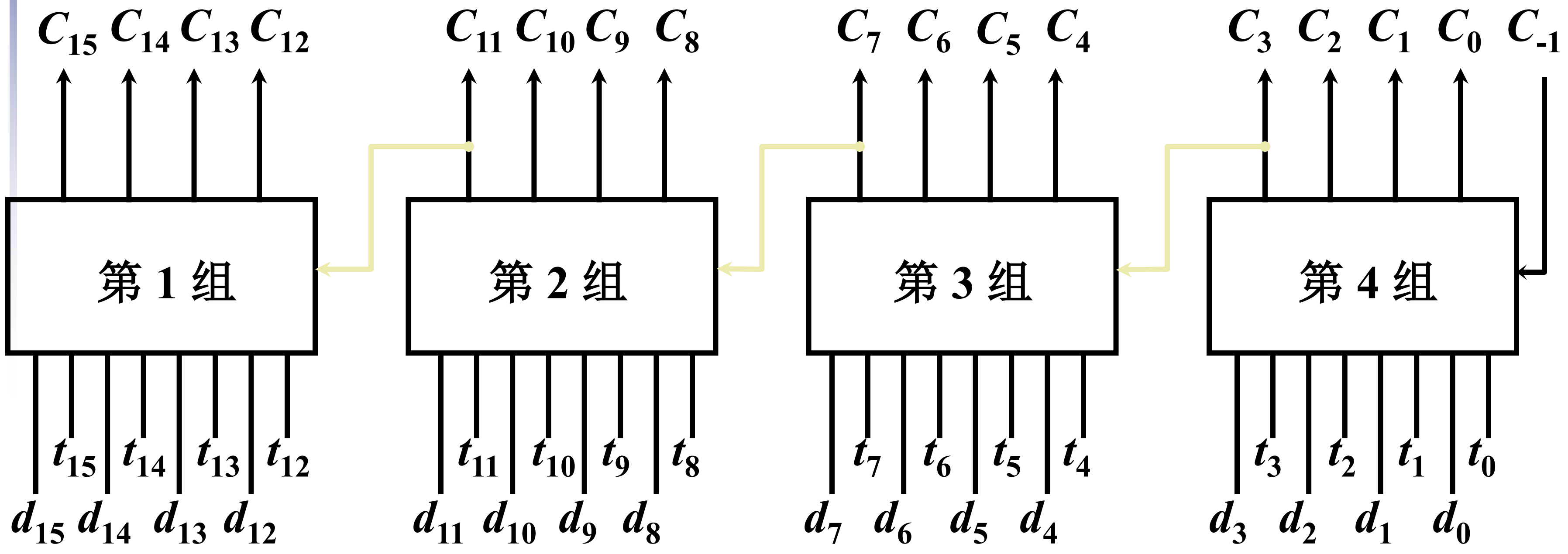
当 $d_i t_i$ 形成后，只需 2.5t_y 产生全部进位

设与或非门的延迟时间为 1.5t_y



(1) 单重分组先行进位链

n 位全加器分若干小组，小组中的进位同时产生，小组与小组之间采用串行进位　以 $n = 16$ 为例



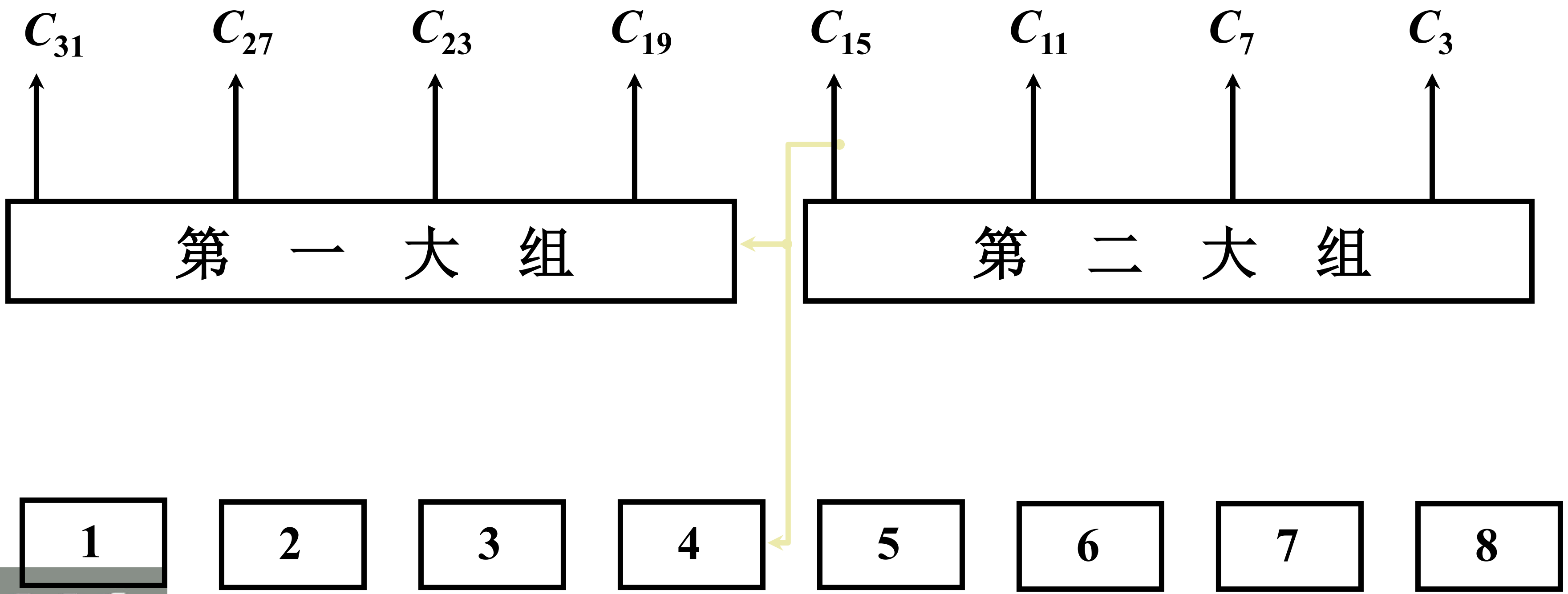
C_{15} C_{14} C_{13} C_{12}
C_{11} C_{10} C_9 C_8
C_7 C_6 C_5 C_4
C_3 C_2 C_1 C_0 C_{-1}
第 1 组
第 2 组
第 3 组
第 4 组
t_{15} t_{14} t_{13} t_{12} t_{11} t_{10} t_9 t_8 t_7 t_6 t_5 t_4 t_3 t_2 t_1 t_0
d_{15} d_{14} d_{13} d_{12} d_{11} d_{10} d_9 d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0

当 $d_i t_i$ 形成后	经 2.5 t_y	产生 $C_3 \sim C_0$
	5 t_y	产生 $C_7 \sim C_4$
	7.5 t_y	产生 $C_{11} \sim C_8$
	1 0 t_y	产生 $C_{15} \sim C_{12}$

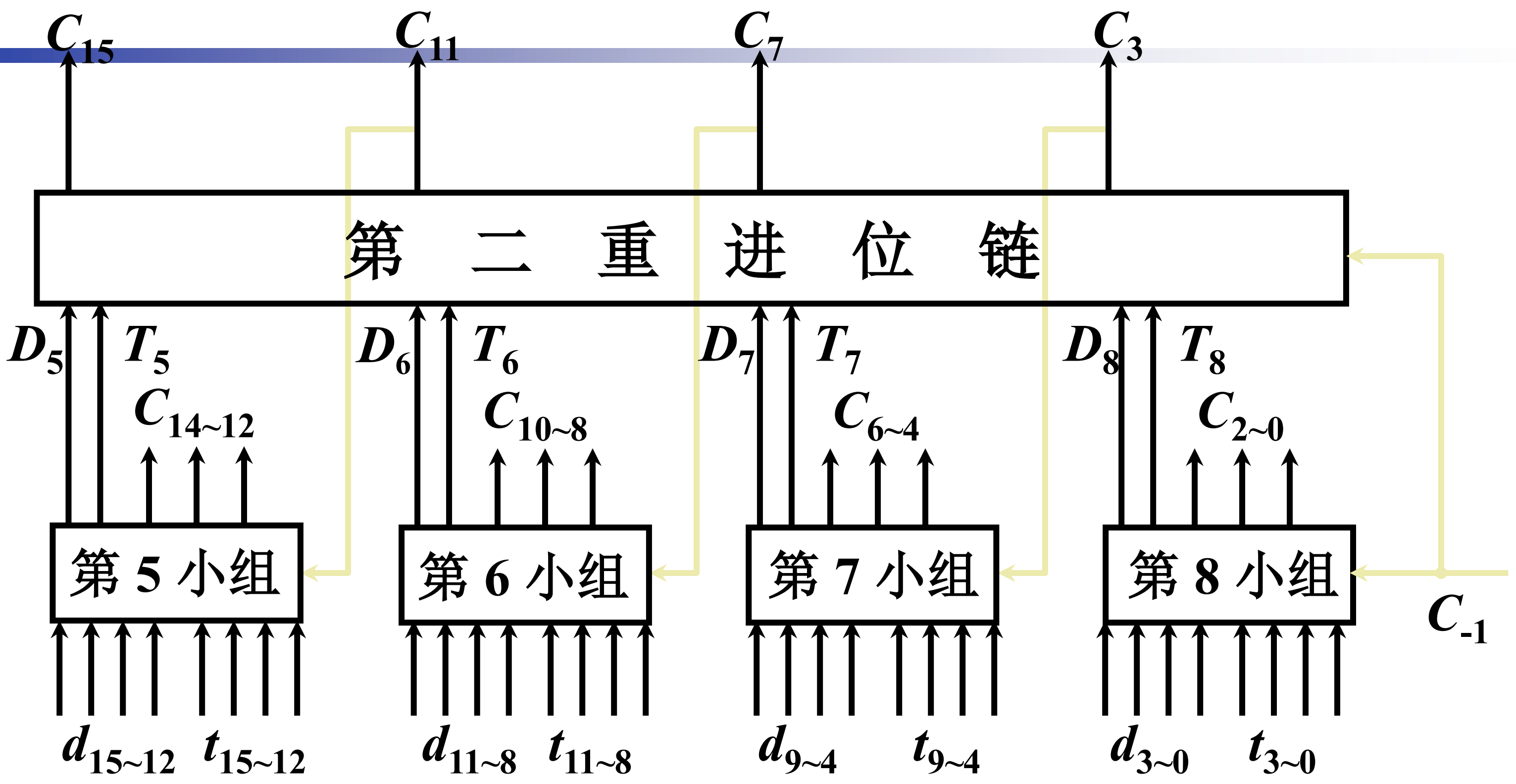
(2) 双重分组先行进位链

n 位全加器分若干大组，大组中又包含若干小组。每个大组中小组的最高位进位同时产生。大组与大组之间采用串行进位。

以 $n = 32$ 为例



(6) n =16 双重分组跳跃进位链



当 $d_i t_i$ 和 C_{-1} 形成后　经 2.5 t_y　产生 C_2、C_1、C_0、$D_5 \sim D_8$、$T_5 \sim T_8$

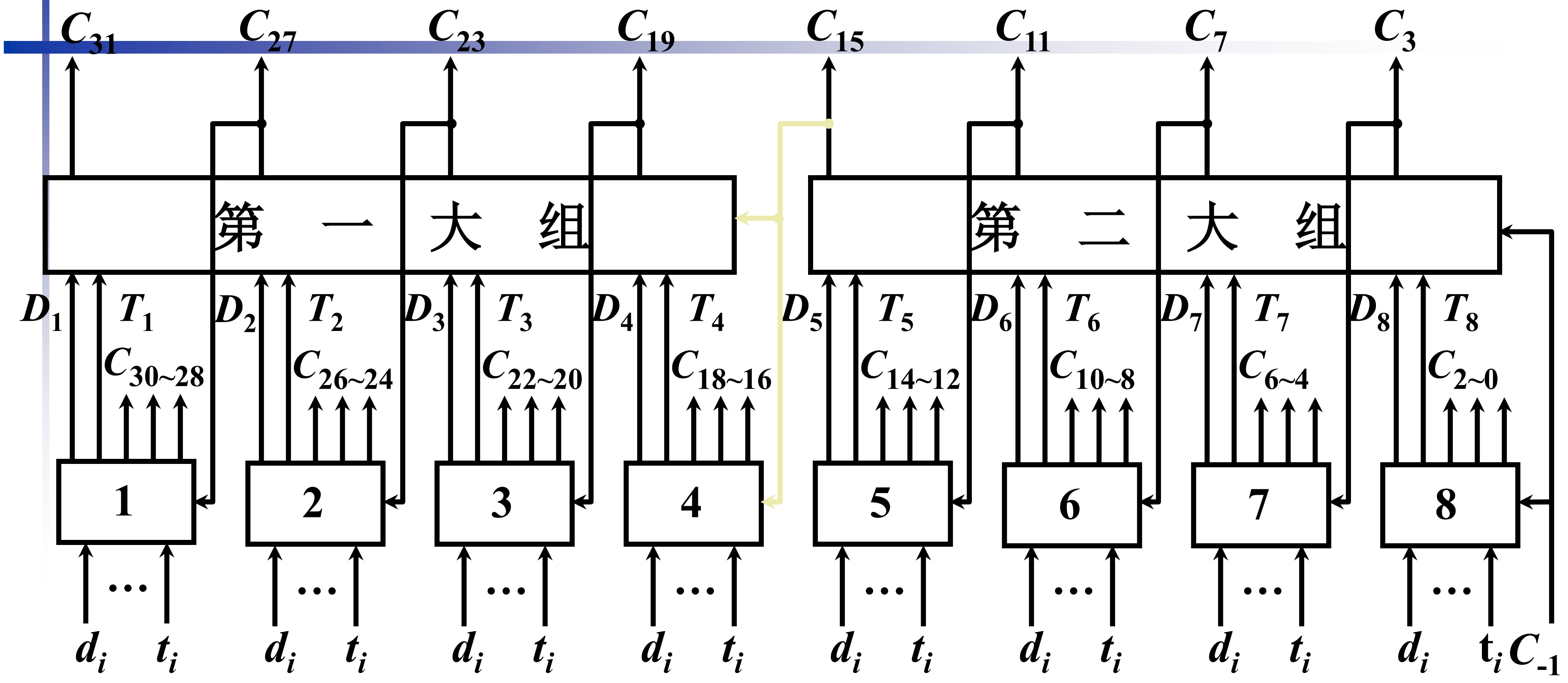
经　5 t_y　产生 C_{15}、C_{11}、C_7、C_3

经 7.5 t_y　产生 C_{14}~C_{12}、C_{10}~C_8、C_6~C_4

串行进位链　经 3 2 t_y　产生　全部进位

单重分组跳跃进位链　经 1 0 t_y　产生　全部进位

(7) n =32 双重分组跳跃进位链



C_{31} C_{27} C_{23} C_{19} C_{15} C_{11} C_7 C_3
第 一 大 组
第 二 大 组
D_1 T_1 D_2 T_2 D_3 T_3 D_4 T_4 D_5 T_5 D_6 T_6 D_7 T_7 D_8 T_8
$C_{30\sim28}$ $C_{26\sim24}$ $C_{22\sim20}$ $C_{18\sim16}$ $C_{14\sim12}$ $C_{10\sim8}$ $C_{6\sim4}$ $C_{2\sim0}$
1 2 3 4 5 6 7 8
d_i … t_i
C_{-1}

当 $d_i t_i$形成后　经 2.5 t_y　产生 C_2、C_1、C_0、$D_1 \sim D_8$、$T_1 \sim T_8$

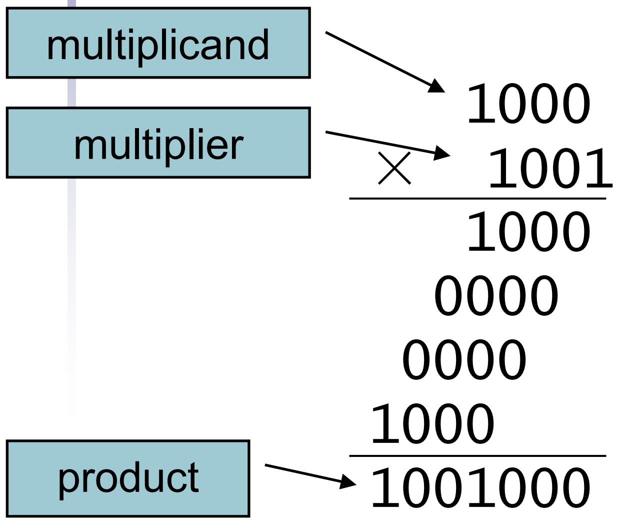
5 t_y　产生 C_{15}、C_{11}、C_7、C_3

7.5 t_y　产生 C_{18}~C_{16}、C_{14}~C_{12}、C_{10}~C_8、C_6~C_4
C_{31}、C_{27}、C_{23}、C_{19}

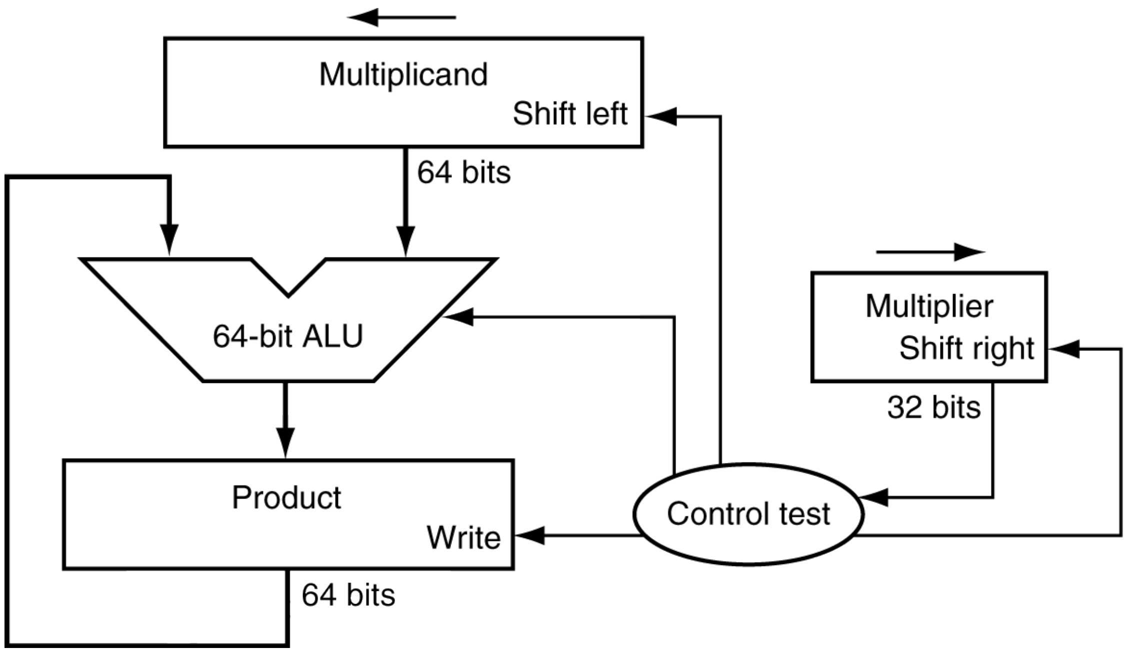
1 0 t_y　产生 C_{30}~C_{28}、C_{26}~C_{24}、C_{22}~C_{20}

Multiplication

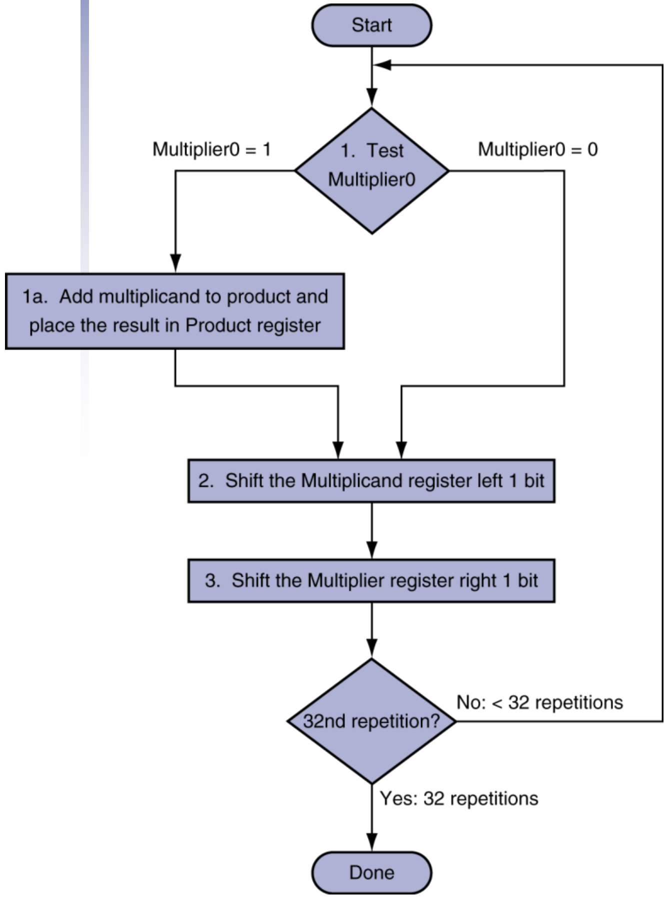
- Start with long-multiplication approach

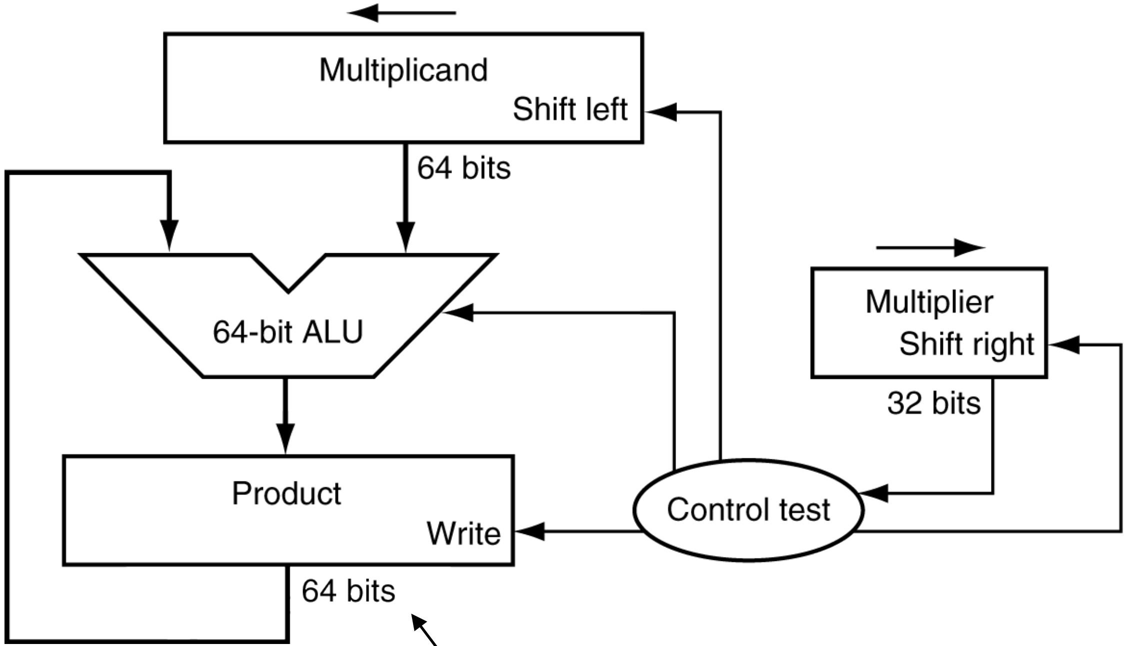


Length of product is the sum of operand lengths



Multiplication Hardware

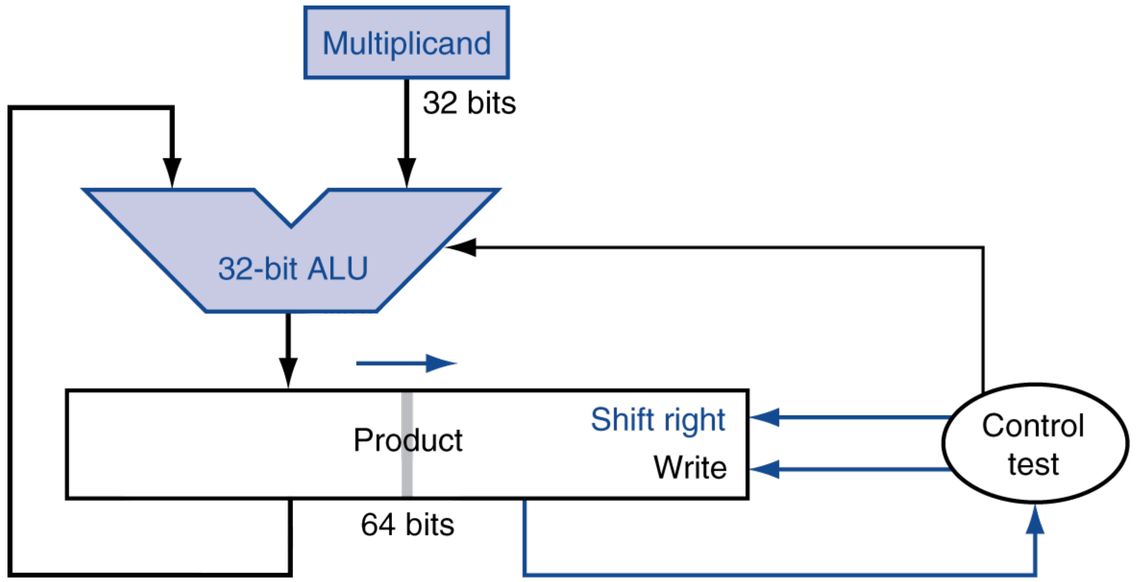

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



Initially 0

Optimized Multiplier

- Perform steps in parallel: add/shift



- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

例题 • 乘法算法

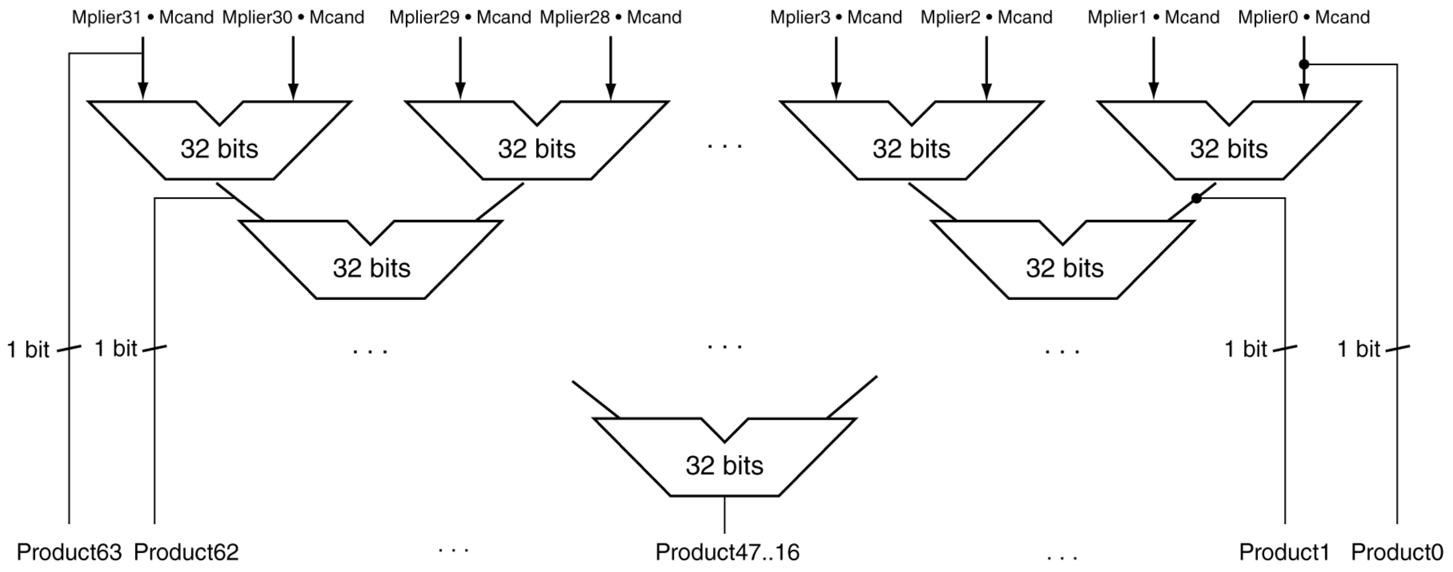
为了节省空间，使用的是 4 位长的数，计算 $2_{10} \times 3_{10}$，或 $0010_2 \times 0011_2$ 的积。

迭代次数	步骤	乘数	被乘数	乘积
0	初始值	001①	0000 0010	0000 0000
1	1a：1⇒乘积=乘积+被乘数	0011	0000 0010	**0000 0010**
	2：左移被乘数	0011	**0000 0100**	0000 0010
	3：右移乘数	**000①**	0000 0100	0000 0010
2	1a：1⇒乘积=乘积+被乘数	0001	0000 0100	**0000 0110**
	2：左移被乘数	0001	**0000 1000**	0000 0110
	3：右移乘数	**000⓪**	0000 1000	0000 0110
3	1：0⇒无操作	0000	0000 1000	0000 0110
	2：左移被乘数	0000	**0001 0000**	0000 0110
	3：右移乘数	**000⓪**	0001 0000	0000 0110
4	1：0⇒无操作	0000	0001 0000	0000 0110
	2：左移被乘数	0000	**0010 0000**	0000 0110
	3：右移乘数	**0000**	0010 0000	0000 0110

图 3-6　使用图 3-4 中算法的乘法例子。圆圈圈起来的是下一步需要检测的位

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff

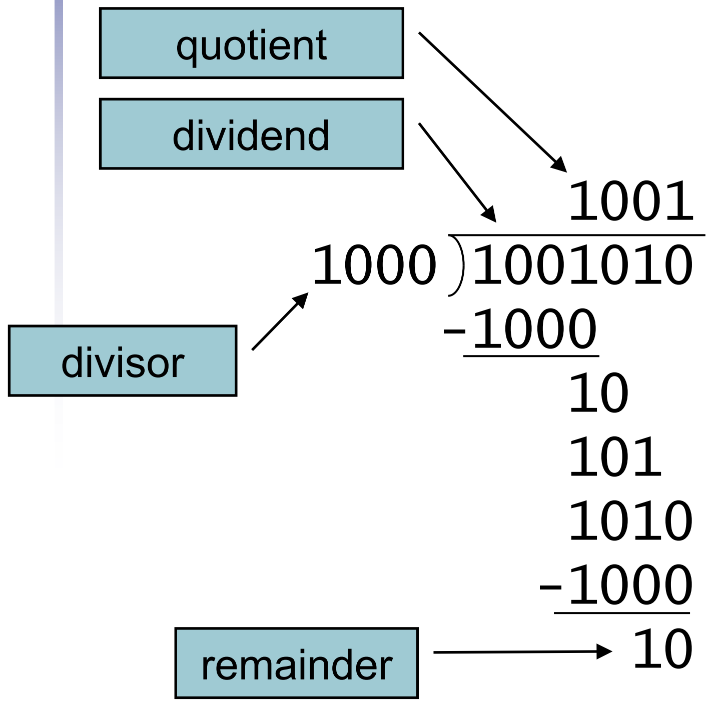


- Can be pipelined
  - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

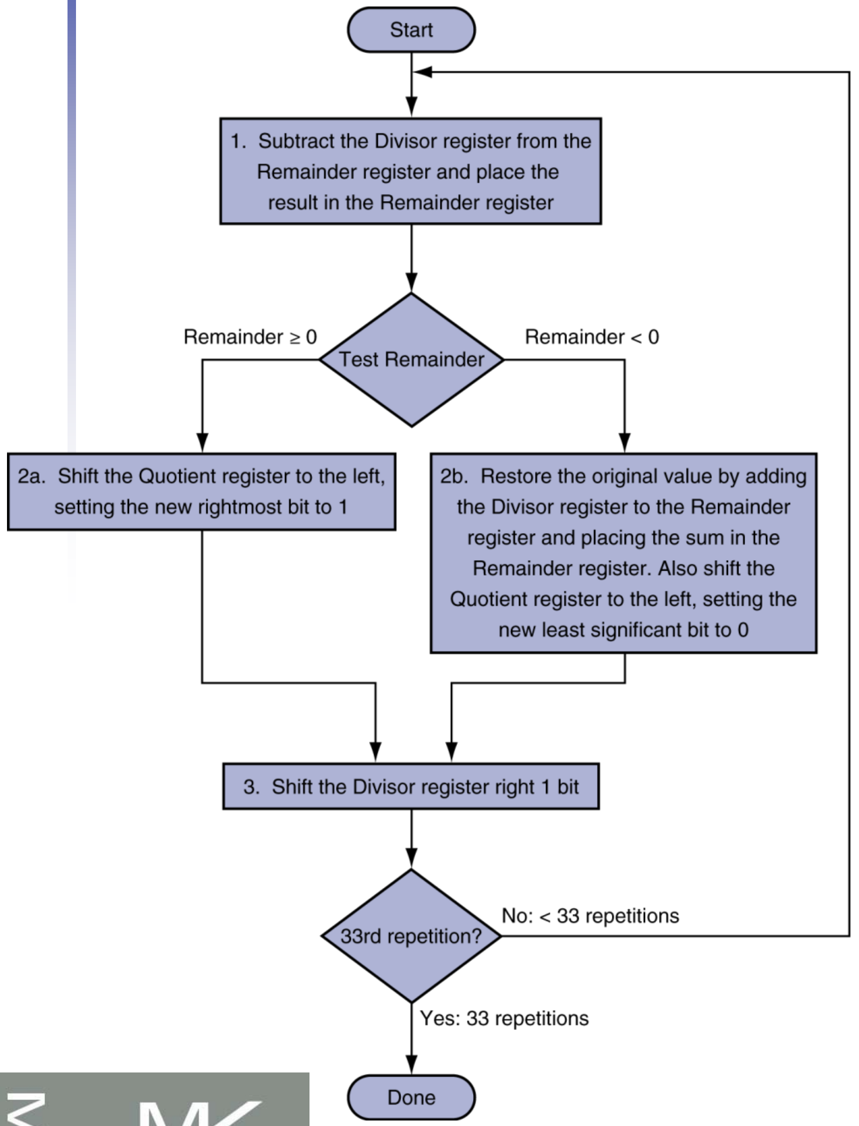
Division



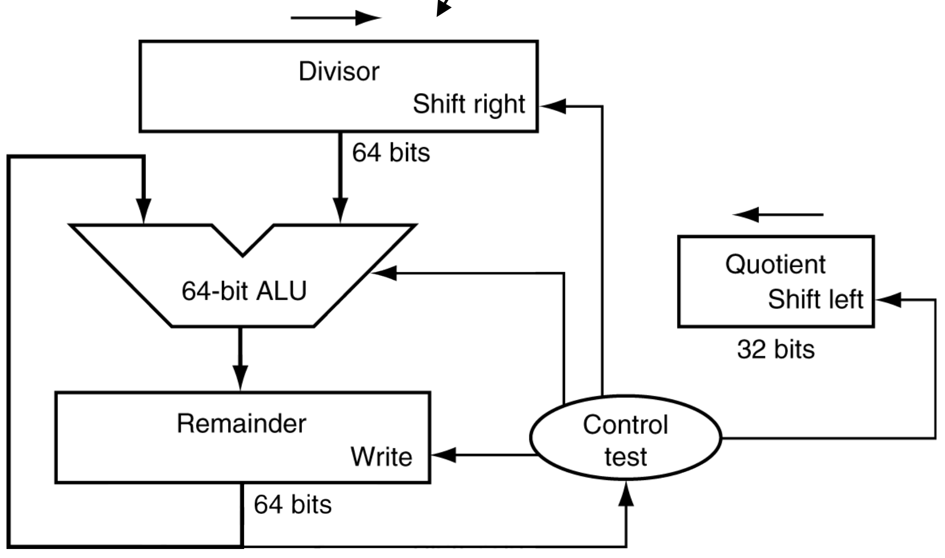
n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware



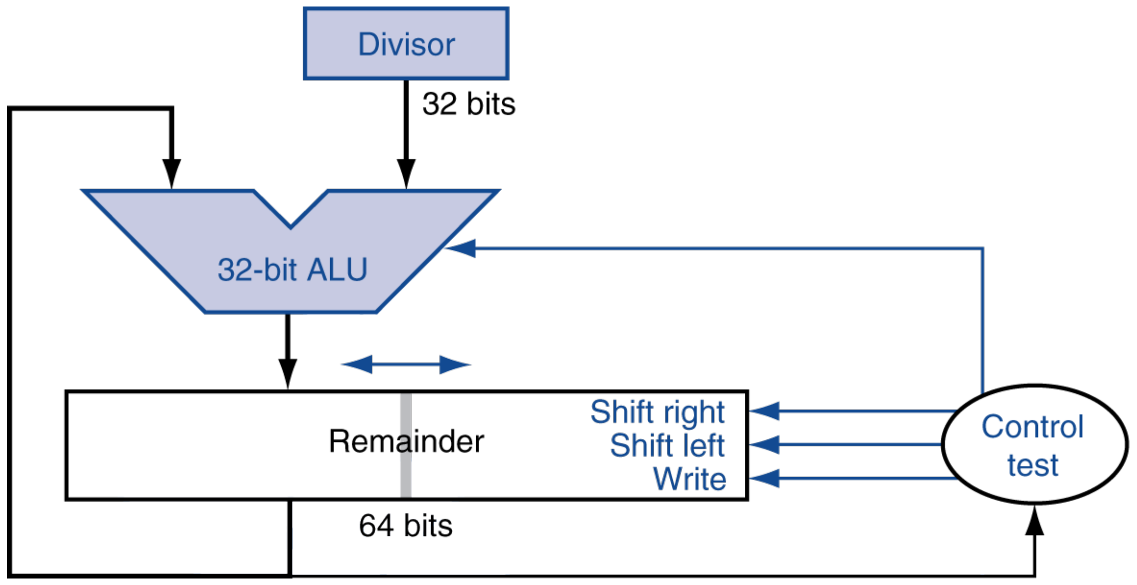
Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



Initially divisor in left half

Initially dividend

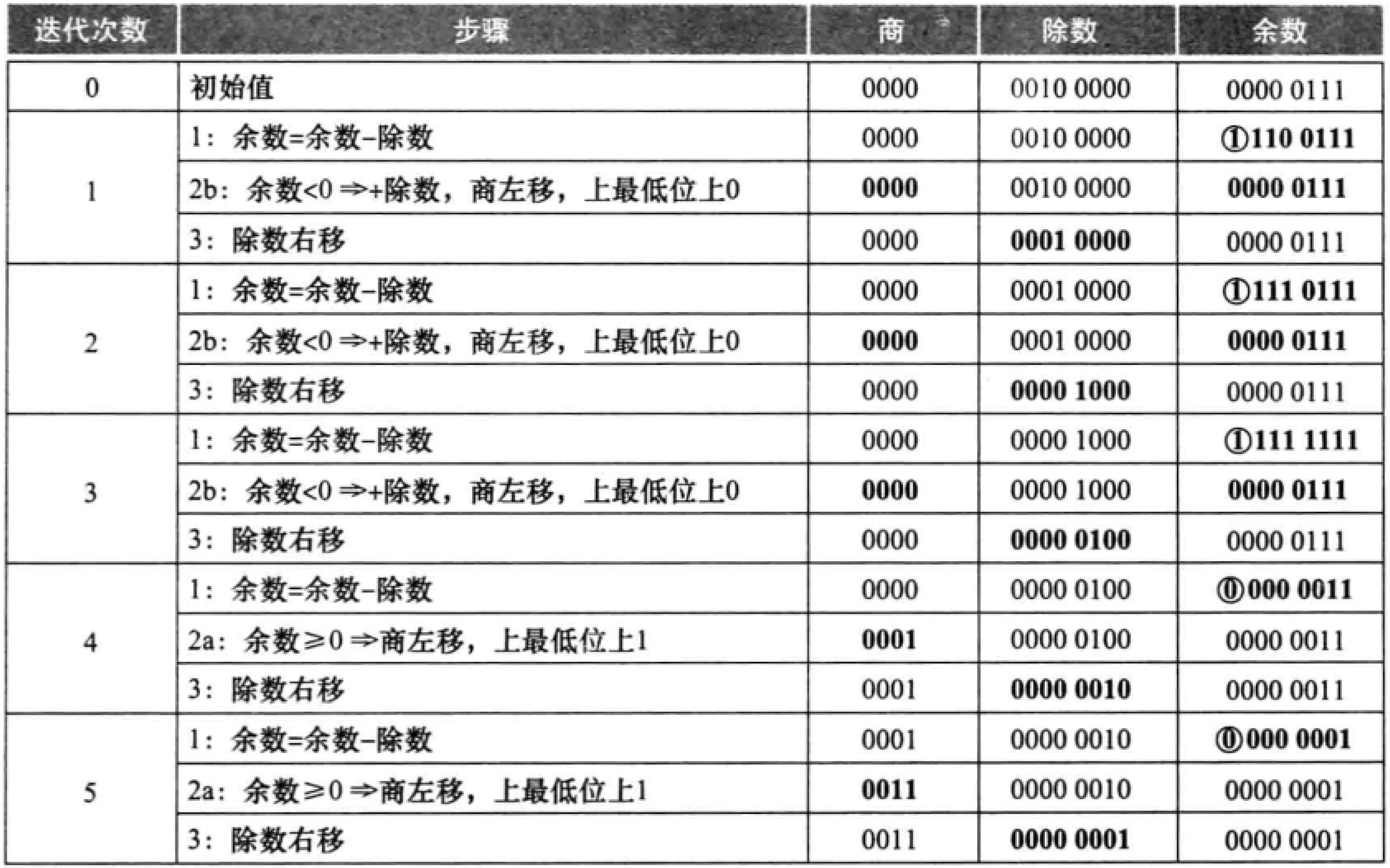
Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

例题・除法算法

为了节省篇幅，我们使用 4 位的数据。计算 7_{10}除以 2_{10}，即 0000 0111_2除以 0010_2。



迭代次数	步骤	商	除数	余数
0	初始值	0000	0010 0000	0000 0111
1	1：余数=余数-除数	0000	0010 0000	**①110 0111**
	2b：余数<0 ⇒+除数，商左移，上最低位上0	**0000**	0010 0000	**0000 0111**
	3：除数右移	0000	**0001 0000**	0000 0111
2	1：余数=余数-除数	0000	0001 0000	**①111 0111**
	2b：余数<0 ⇒+除数，商左移，上最低位上0	**0000**	0001 0000	**0000 0111**
	3：除数右移	0000	**0000 1000**	0000 0111
3	1：余数=余数-除数	0000	0000 1000	**①111 1111**
	2b：余数<0 ⇒+除数，商左移，上最低位上0	**0000**	0000 1000	**0000 0111**
	3：除数右移	0000	**0000 0100**	0000 0111
4	1：余数=余数-除数	0000	0000 0100	**⓪000 0011**
	2a：余数≥0 ⇒商左移，上最低位上1	**0001**	0000 0100	0000 0011
	3：除数右移	0001	**0000 0010**	0000 0011
5	1：余数=余数-除数	0001	0000 0010	**⓪000 0001**
	2a：余数≥0 ⇒商左移，上最低位上1	**0011**	0000 0010	0000 0001
	3：除数右移	0011	**0000 0001**	0000 0001

图 3-10　除法的例子，采用图 3-9 中的算法。图中圈起来的位用于决定下一步的操作

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT devision) generate multiple quotient bits per step
  - Still require multiple steps

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt  /  divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

(2) 不恢复余数法（加减交替法）

- 恢复余数法运算规则

余数 $R_i>0$ 上商 “1”， $2R_i - y^*$

余数 $R_i<0$ 上商 “0”， $R_i + y^*$ 恢复余数

$$2(R_i+y^*) - y^* = 2R_i + y^*$$

- 不恢复余数法运算规则

上商 “1” $2R_i - y^*$

上商 “0” $2R_i + y^*$

加减交替

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = 10000001_2 = 129
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

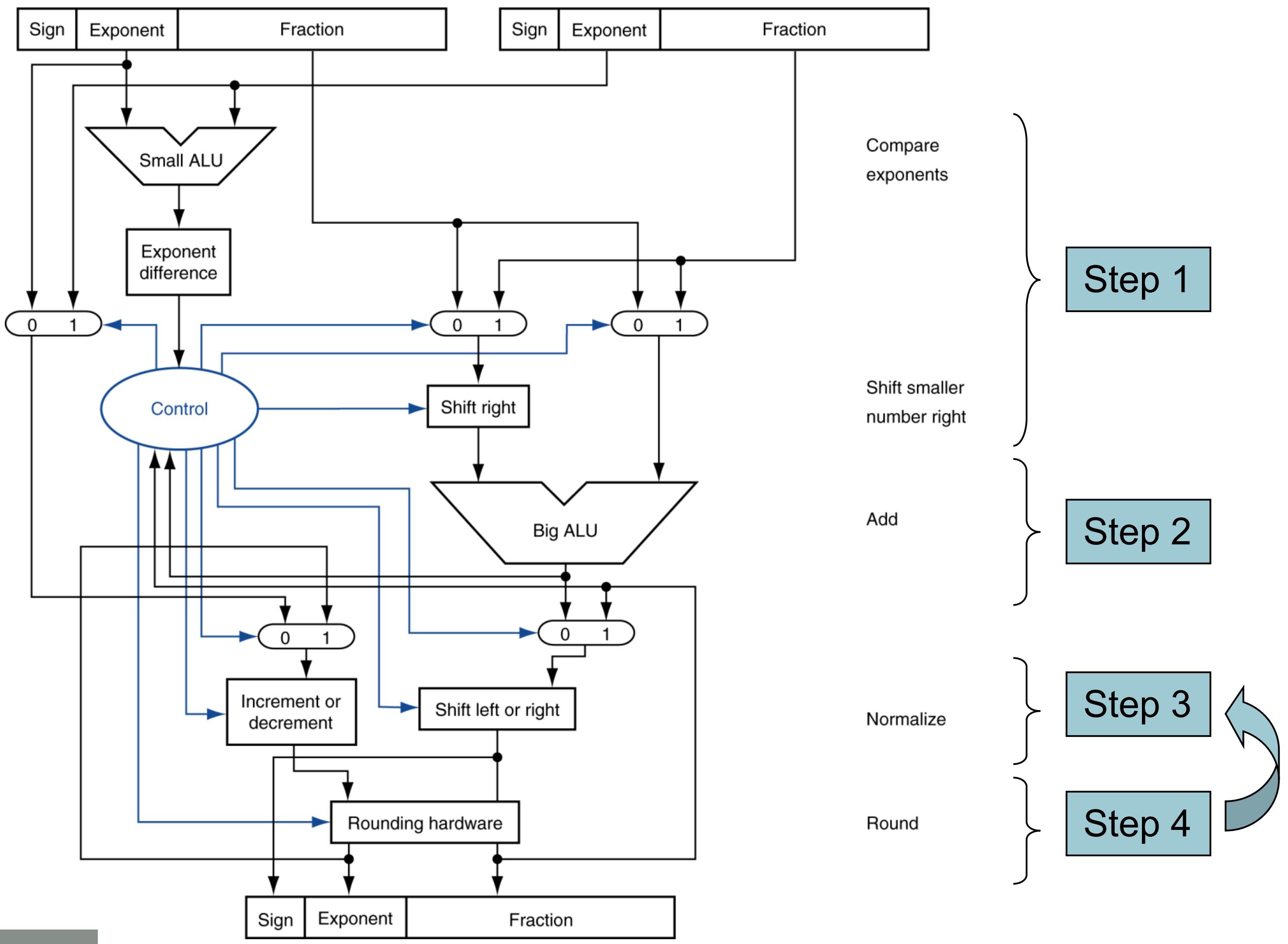
Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: \$f0, \$f1, … \$f31
  - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, …
    - Release 2 of MIPs ISA supports 32 $\times$ 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is eq, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

FP Example: ° F to ° C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

FP Example: Array Multiplication

- $X = X + Y \times Z$
  - All 32×32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double x[][],
              double y[][], double z[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        x[i][j] = x[i][j]
                   + y[i][k] * z[k][j];
}
```

  - Addresses of x, y, z in \$a0, \$a1, \$a2, and i, j, k in \$s0, \$s1, \$s2

FP Example: Array Multiplication

- MIPS code:

```
    li   $t1, 32         # $t1 = 32 (row size/loop end)
    li   $s0, 0          # i = 0; initialize 1st for loop
L1: li   $s1, 0          # j = 0; restart 2nd for loop
L2: li   $s2, 0          # k = 0; restart 3rd for loop
    sll  $t2, $s0, 5     # $t2 = i * 32 (size of row of x)
    addu $t2, $t2, $s1 # $t2 = i * size(row) + j
    sll  $t2, $t2, 3     # $t2 = byte offset of [i][j]
    addu $t2, $a0, $t2 # $t2 = byte address of x[i][j]
    l.d  $f4, 0($t2)     # $f4 = 8 bytes of x[i][j]
L3: sll  $t0, $s2, 5     # $t0 = k * 32 (size of row of z)
    addu $t0, $t0, $s1 # $t0 = k * size(row) + j
    sll  $t0, $t0, 3     # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
    l.d  $f16, 0($t0)    # $f16 = 8 bytes of z[k][j]
```

…

FP Example: Array Multiplication

```
    …
    sll  $t0, $s0, 5        # $t0 = i*32 (size of row of y)
    addu  $t0, $t0, $s2     # $t0 = i*size(row) + k
    sll   $t0, $t0, 3       # $t0 = byte offset of [i][k]
    addu  $t0, $a1, $t0     # $t0 = byte address of y[i][k]
    l.d   $f18, 0($t0)      # $f18 = 8 bytes of y[i][k]
    mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
    add.d $f4, $f4, $f16    # f4=x[i][j] + y[i][k]*z[k][j]
    addiu $s2, $s2, 1       # $k k + 1
    bne   $s2, $t1, L3      # if (k != 32) go to L3
    s.d   $f4, 0($t2)       # x[i][j] = $f4
    addiu $s1, $s1, 1       # $j = j + 1
    bne   $s1, $t1, L2      # if (j != 32) go to L2
    addiu $s0, $s0, 1       # $i = i + 1
    bne   $s0, $t1, L1      # if (i != 32) go to L1
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
`FILD mem/ST(i)`	`FIADDP mem/ST(i)`	`FICOMP`	`FPATAN`
`FISTP mem/ST(i)`	`FISUBRP mem/ST(i)`	`FIUCOMP`	`F2XMI`
`FLDPI`	`FIMULP mem/ST(i)`	`FSTSW AX/mem`	`FCOS`
`FLD1`	`FIDIVRP mem/ST(i)`		`FPTAN`
`FLDZ`	`FSQRT`		`FPREM`
	`FABS`		`FPSIN`
	`FRNDINT`		`FYL2X`

- Optional variations
  - `I`: integer operand
  - `P`: pop operand from stack
  - `R`: reverse operand order
  - But not all combinations allowed

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - Instructions operate on them simultaneously
    - <u>S</u>ingle-<u>I</u>nstruction <u>M</u>ultiple-<u>D</u>ata

Matrix Multiply

- Unoptimized code:

```
void dgemm (int n, double* A, double* B, double* C)
{
 for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j)
   {
    double cij = C[i+j*n]; /* cij = C[i][j] */
    for(int k = 0; k < n; k++ )
     cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij; /* C[i][j] = cij */
   }
}
```

Matrix Multiply

- x86 assembly code:

```
vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
mov %rsi,%rcx         # register %rcx = %rsi
xor %eax,%eax         # register %eax = 0
vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
add %r9,%rcx          # register %rcx = %rcx + %r9
vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
add $0x1,%rax         # register %rax = %rax + 1
cmp %eax,%edi         # compare %eax to %edi
vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
jg 30 <dgemm+0x30>    # jump if %eax > %edi
add $0x1,%r11d        # register %r11 = %r11 + 1
vmovsd %xmm0,(%r10)   # Store %xmm0 into C element
```

Matrix Multiply

- Optimized C code:

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
             _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
             _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
 }
```

Matrix Multiply

- Optimized x86 assembly code:

```
vmovapd (%r11),%ymm0        # Load 4 elements of C into %ymm0
mov %rbx,%rcx               # register %rcx = %rbx
xor %eax,%eax               # register %eax = 0
vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
add $0x8,%rax               # register %rax = %rax + 8
vmulpd (%rcx),%ymm1,%ymm1   # Parallel mul %ymm1,4 A elements
add %r9,%rcx                # register %rcx = %rcx + %r9
cmp %r10,%rax               # compare %r10 to %rax
vaddpd %ymm1,%ymm0,%ymm0    # Parallel add %ymm1, %ymm0
jne 50 <dgemm+0x50>         # jump if not %r10 != %rax
add $0x1,%esi               # register % esi = % esi + 1
vmovapd %ymm0,(%r11)        # Store %ymm0 into 4 C elements
```

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
  - c.f. 11111011_2 >>> 2 = 00111110_2 = +62

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

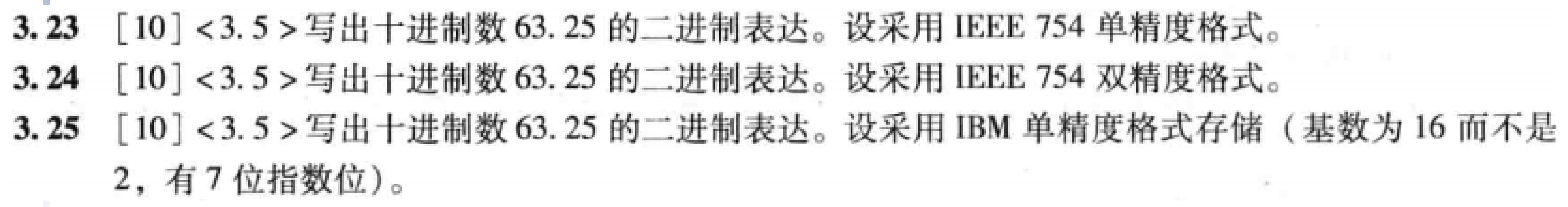
Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

Concluding Remarks



3.23 [10] <3.5>写出十进制数 63.25 的二进制表达。设采用 IEEE 754 单精度格式。

3.24 [10] <3.5>写出十进制数 63.25 的二进制表达。设采用 IEEE 754 双精度格式。

3.25 [10] <3.5>写出十进制数 63.25 的二进制表达。设采用 IBM 单精度格式存储（基数为 16 而不是 2，有 7 位指数位）。